Rover State Monitoring
for Development and Operation

Requirements and Design Concept

Why Monitor Internal State?

Development

Recording component internal state is crucial to software development and testing, particularly debugging

Internal state information is needed for analysis of timing and function to improve performance

Operation

During operation state can be interpreted for health assessment, fault recovery, and performance optimization

Telemetry of state information (engineering data) is needed for situational awareness and operational planning

But Why?

Inevitable

- CLARAty is formulating a new architecture and developing a potentially large and open code base
- Many developers and much code building must ultimately produce lean, stable rover software
- Developers will need tools and will either use services provided by CLARAty or will use their own scheme of printfs, messages, and log files
- It could get very messy and/or computationally expensive
- Overall it may be more effective to develop a uniform, stable, efficient, extensible state monitoring scheme

Objective

Develop a CLARAty component to enable internal state extraction, transmission and monitoring for use during rover development and operation

Using Internal State

State Extraction

Collecting internal state information from source components (instrumenting components, logging state)

State Transmission

Transmitting state information to destination components

State Recording

Recording a time-referenced sequence of state including events, faults, and description

State Monitoring

Run-time use of internal state for operation

Requirements

Standardized

State monitoring must be object oriented, make use of CLARAty base classes and tools, and conform to architecture decomposition

Scalable

Large numbers of components each with internal state that must be monitored during development and operation

Robust

Components will crash, hang,and adversely effect other components directly but state monitoring should not have indirect affect

Timing

State monitoring must be able to time synchronize internal state information for event traceablilty

Getting timing correct is key to useability

Requirements

Distributed

State monitoring will necessarily be distributed so the transmission scheme must support broad distribution

Efficient

State monitoring must minimize impact on the performance of the instrumented objects (for example it should not hog thread cycles)

State monitoring must minimize impact on total system performance (for example it should not clog the bus with message traffic)

Flexible

State monitoring has many potential uses from producing customized log files to dynamically adjusting component priorities to optimize efficiency so flexibility is important

Portable

State monitoring must be OS/HW independent or easily portable

Use of abstractions, like ACE middleware, are desirable

Design Concept

Client-Server Architecture

State Server

- Accepts connections from clients
- Exchanges state information

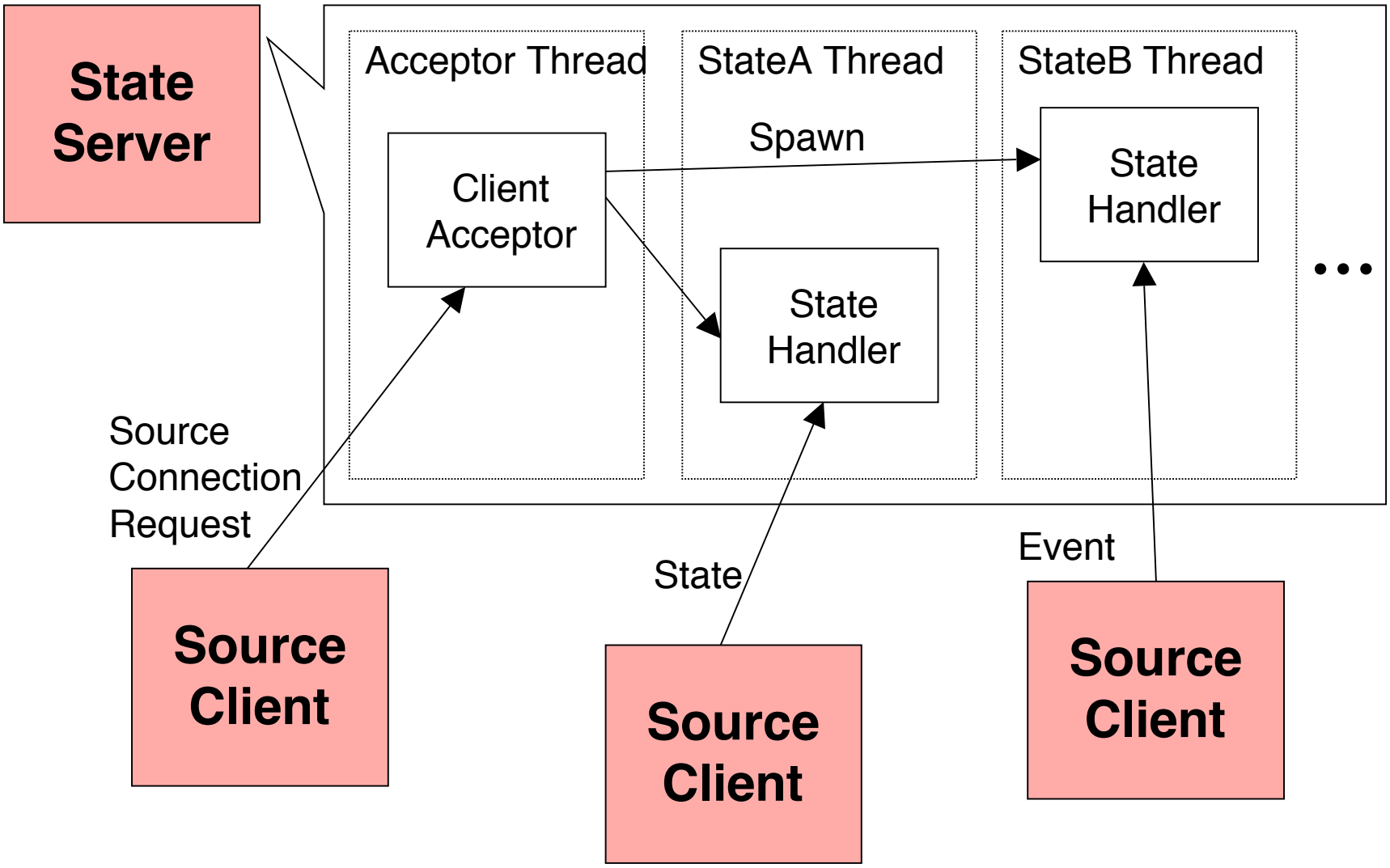
State Sources (Source Clients)

- Connect to server
- Extract internal state
- Transmit to server

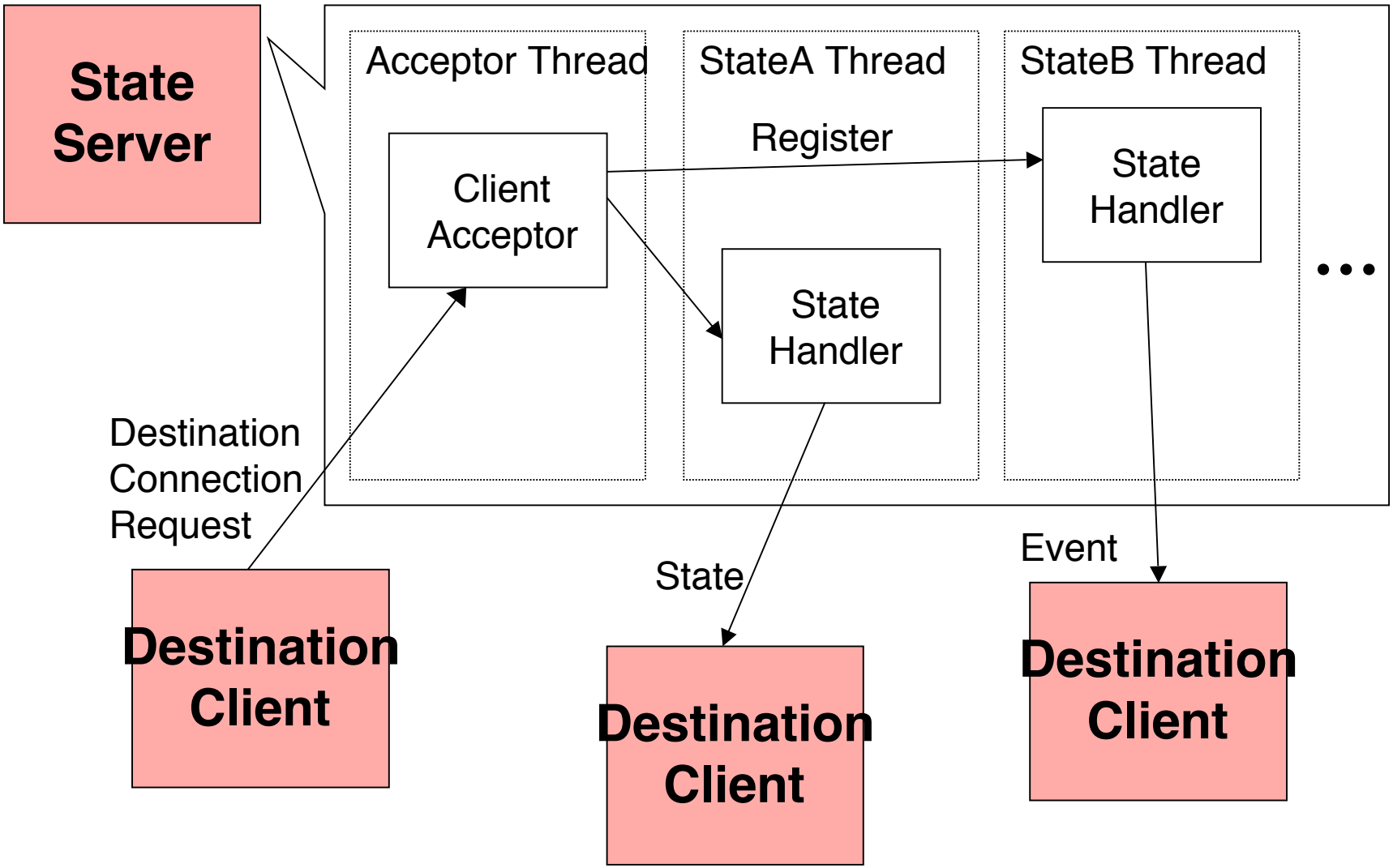
State Destination (Destination Clients)

- Connect to server
- Register desired state
- Receive state stream from server

State Server and Source Clients



State Server and Destination Clients



State Clients

Source Clients

Modules with monitored state

- Almost Every Module
- Sensor Interfaces, Controllers, Navigators, Planners
- Destination Clients

Destination Clients

Modules that utilize state information

- Telemetry Manager
- Health Monitor
- System Executive
- Performance Analyzer

State Extraction

Desired Properties

1. Minimal code modification
2. Minimal computational impact (use OS)
3. Non-blocking

State Extraction

Possible Approaches

Methods are specifically invoked to capture state

Monitored variables are registered and then sampled

Objects are instrumented and state is passively extracted

State Extraction - Invocation

State is extracted by invoking logging methods

- Object data logged by invoking an extraction method
- Developer must invoke methods for state transition (not automatic for each state transition) or other events
- Enables log-by-event

Implementation concepts:

- Use included class to extract state data and transmit
- Could also use to extract other information such as events, signals, triggers, process status, data from other components, even debugging messages
- (Similar to rlog and rlogEvent approach)

State Extraction - Sampling

State is extracted by registering variables

- Object data to be extracted is registered
- Extraction occurs when method is invoked to sample all the registered data
- Enables log-by-time

Implementation concepts:

- Use included class to register object internal state variables. Might get messy with complex data hiding (works great with C-style global variables)
- Method for harvesting the data tied to event or timing loop
- Similar to rlogRegisterVariable and rlogFlushChanges approach

State Extraction - Instrumentation

State is “instrumented”

Specific object data is implemented so that its value can be extracted at run-time.

Developer is responsible for instrumenting state by designating the instrumentation class but extraction is otherwise transparent

Implementation concepts:

Use templated class to overload the assignment operator for instrumented state.

Assignment operator also extracts state.

Implementation possible for primitive types but may be problematic for complex data types.

State Transmission

Desired Properties

1. Minimized computational cost to client (use OS)
2. Minimized bus traffic (multi-cast)
3. Secure communication
4. Minimize delays/delays do not effect clients
5. Temporal order maintained
6. Minimal queuing
7. Dynamic redirection
8. Non-blocking

State Transmission

Implementation Concepts

- Shared memory - very efficient but not extensible/portable
- Virtual shared memory - management and portability issues
- Sockets - tricky to establish and manage, need wrapper
- Mailboxes/messages - management and portability issues
- Rlog - nice plug-in concept but client impact and state transmission (via non-portable sockets or IPC) issues
- ACE - communication middleware, encapsulation for distributed transmission

State Transmission

Communication using ACE middleware

- Based on ACE thread-per-connection server model
- ACE sockets encapsulate everything from addressing, to connecting, to data (de)marshalling
- Non-blocking connections
- Scatter-write and gather-read for efficiency
- Robust to client disconnect

Connection per state type

- State server establishes state handler per state type
- State source client extract and transmit
- State destination clients register and receive

Time Recording

Recording state time and relating times from various distributed components is essential

- Each connection from a client to the State Server is established with identification of host process and local time information
- Local time is available through ACE_getlocaltime a portable method
- State Server has a information to determine time offsets between all client logged threads.
- Time offsets can be used to produce time corrected state and event sequences

State Logs

Created by the State Server

- Individual component logs (each associated with transmission thread)
- Master log corrected for clock offsets of each state client

Created by State Destination Clients

- Telemetry Manager can log filtered state
- Destination off-board can record state to off-load storage requirements

State Analysis

Desired Properties

1. Obtain state from multiple sources
2. Resequence state by time
3. Vary verbosity
4. Vary sample rates

Telemetry Manager Concept

On-board Telemetry Manager has broad subscription for state from State Server

- Comprehensive state information received from State Server can be filtered and prioritized for downlink
- Detailed engineering can be buffered for optional downlink
- Fault data can be buffered for optional downlink
- Processed state unicast to off-board Telemetry Manager

Off-board Telemetry Manager receives state and multicasts to destination components

- ACE multicast supports efficient, secure communication
- Operator interfaces, analysis, planners, public outreach

Health Monitor Concept

On-board Heath Monitor analyzes state

Fault Detection (Simple Faults)

- Component triggered faults
- Component process condition
- Individual parameters out-of-bound
- Composite parameters out-of-bound

Fault Detection (Complex Faults)

- Diverse state and inference (high-centering fault)

Fault Forecasting

- Detecting trends that lead to faults
- Predicting the likelihood of a fault

Conclusions

State monitoring is needed for development and operation

CLARAty components should be instrumented to extract internal state

ACE middleware would support a multithreaded State Server to collect, record, and distribute state

CLARAty components such as Rover Executive, Health Monitor, Telemetry Manager would utilize the State Server